The Thesis Committee for Anisa Aziz Qazi
Certifies that this is the approved version of the following thesis:

Hardware Accelerator for ALICE ITS Cluster Finder

APPROVED BY

CO-SUPERVISING COMMITTEE:

Jacob Abraham, Co-Supervisor

Christina Markert, Co-Supervisor

Hardware Accelerator for ALICE ITS Cluster Finder

by

Anisa Aziz Qazi

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

Acknowledgments

I would like to thank my advisors Dr. Jacob Abraham and Dr. Christina Markert for giving me the opportunity of being a part of CERN and contributing to the ALICE project. Their guidance and support throughout the course of this project was invaluable. I would also like to thank Dr. Joachim Schambach for being a mentor and helping me with my work on a day to day basis. I appreciate all the inputs I got from him, which helped me progress with this work in a streamlined fashion. I would also like to thank Ruben Shahoyan, Filippo Costa and Jozsef Imrek for their timely help.

Hardware Accelerator for ALICE ITS Cluster Finder

Anisa Aziz Qazi, M.S.E.
The University of Texas at Austin, 2018

Co-Supervisors: Jacob Abraham
Christina Markert

An integral part of the upgrade to the Inner Tracking System (ITS) of the ALICE detector is to support increased readout rates of the charged particles resulting due to increased interaction rate of 50kHz in Pb-Pb collisions at the Large Hadron Collider (LHC). A major task of the ITS readout system is to compress the data and store it in the mass storage system for later analysis. The first step of data compression involves cluster finding on the pixel data received from ALPIDE sensors followed by Huffman compression. In this Thesis, we evaluate the resource requirements for implementing cluster finding on the Arria 10 FPGAs which are an integral part of the ITS readout system, in an attempt to reduce the computing nodes needed on the First Level Processors (FLPs) and also to speed up the processing. We present a hardware implementation of a single pass Connected Component Labeling algorithm. A special linked list based merger table that ensures a constant worst case latency for chained label mergers independent of their length is proposed. For retrieving the shapeIDs, pixels are segregated into clusters on-the-fly without the need

to store labeled pixels in memory. Verilog code implementing this design has been written, a testbench for functional verification has been developed, and the design has been synthesized.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

ALICE (A Large Ion Collider Experiment) [1], is one of the four larger experiments at the CERN LHC (Large Hadron Collider). It is designed to study the physics of strongly interacting matter at extreme energy densities and temperatures, in particular, the Quark-Gluon Plasma (QGP), using high-energy proton-proton, proton-nucleus and nucleus-nucleus collisions (about 7 TeV per nucleon). ALICE is preparing a major upgrade of its apparatus during the LS2 (Long Shutdown 2), currently planned during 2019-2020. It aims at upgrading the detectors, which, combined with a significant increase of luminosity of the LHC, will enhance the physics capabilities of ALICE enormously.

The LHC can presently deliver Pb-Pb collision rates of 8kHz. After the LS2, ALICE detector will be upgraded to cope with an interaction rate of 50kHz (luminosity L = 6 x 10^{27} cm^{-2} s^{-1} [2]) producing a sustained data throughput of 1TB/s from the ALICE detector. These data will be reduced in the online-offline (O^2) computing system by means of online reconstruction and data reduction so that the stream to permanent storage stays below 80GB/s peak with a 20GB/s average [3]. In order to reduce the number of compute

nodes needed in the O^2 system to achieve this task, Altera Arria 10 FPGAs, which are an integral part of the readout system, will be utilized to implement Cluster finding as a first step of online reconstruction to achieve data reduction.

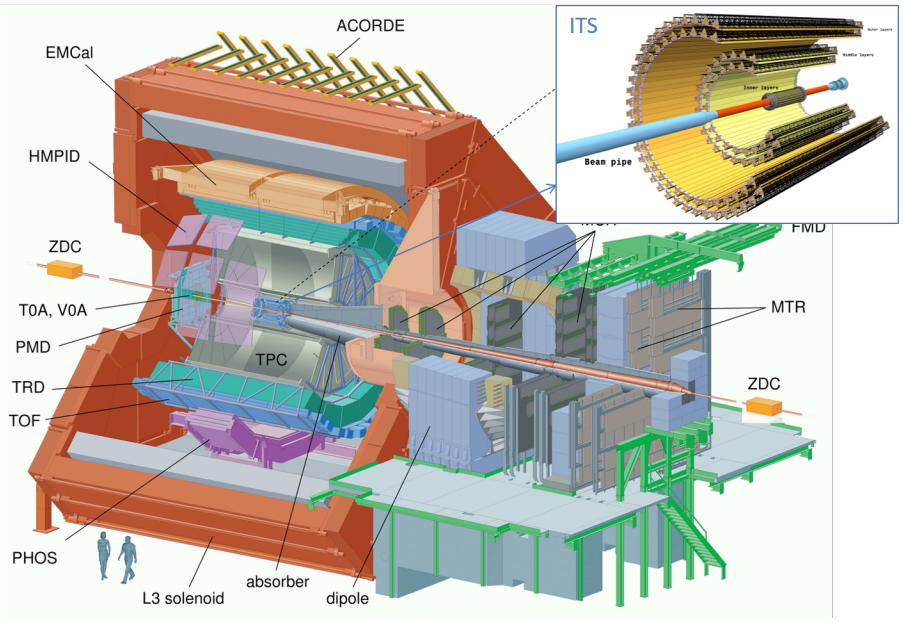

Figure 1.1: The ALICE detector

In this Thesis, we discuss the RTL implementation of the Cluster finding algorithm for the Inner Tracking System (ITS). The ITS consists of multiple layers of pixel detectors surrounding the collision point in the ALICE detector (shown in figure 1.1 taken from [4]). The problem of Cluster finding can be split into 2 major steps: Connected Component Labeling (CCL) and Feature extraction for each component. A classical 2-pass connected component labeling requires storing the complete image in memory. This is not very efficient for FPGA implementations as this may require the use of external

high bandwidth memory, thus, slowing down the processing. In this work, we use a single pass connected component labeling algorithm, reducing the memory requirement and also reducing the latency for corresponding memory accesses. We use a linked list based approach for on-the-fly label merging which takes care of the problem of merger chains. For feature extraction, we propose a shift register based approach to get the shape bits on-the-fly in a pipelined manner without the need to store all the pixels of a component.

The rest of the Thesis is structured as follows. Chapter 2 discusses the construction of the ALICE ITS detector and the complete readout system. Chapter 3 discusses the existing work on Connected Component Labeling and the problems that need to be solved in such a system. In Chapter 4, we present the RTL implementation of our design. In Chapter 5, we discuss the results of our work. We conclude this Thesis and present some ideas for further improvement of this work in Chapter 6.

Chapter 2

ALICE ITS Detector

The LS2 upgrade of the LHC foresees an increased event rate of more than 50kHz of Pb-Pb collisions. This calls for increased resolution and readout capabilities of the Inner Tracking System (ITS). This chapter discusses the capabilities of the current ITS detector, the construction of the new ITS detector and the associated readout system that will be needed for the upgrade.

2.1 The current ITS

The short-living heavy particles travel a very small distance from the collision vertex before decaying. The ITS aims at identifying these phenomena of decay by measuring the location where it occurs.

The current ITS detector consists of 6 layers of silicon detectors: 2 layers of Silicon Pixel Detectors (SPD), 2 layers of Silicon Drift Detectors (SDD) and 2 layers of Silicon Strip Detectors (SSD) as shown in figure 2.1. The pixel size of the current detectors is 50um x 425um; this will become significantly smaller with the upgrade, to increase the resolution of the detector. The readout rate is currently limited to 1 kHz.

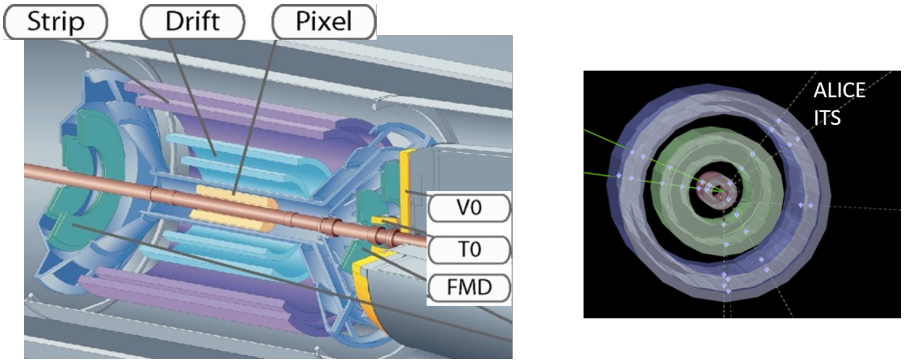

Figure 2.1: The current ITS detector

2.2 Construction of the new ITS detector

The new ITS will be made of 7 layers of monolithic pixel detectors, the details of which will be discussed in the following sections.

2.2.1 The ALPIDE sensor

A custom developed sensor called 'ALPIDE' is the basic detector element used in the ITS. It is based on Monolithic Active Pixel Sensor (MAPS) technology and is implemented in 180nm CMOS technology [5]. It measures 15mm X 30 mm and contains a matrix of 512 x 1024 sensitive pixels (0.5 MPixels). Each pixel is 29.24um x 26.88um. There are a total of ≈24000 sensors in the ITS detector giving it a resolution of 12.5GPixels.

The maximum occupancy[1] of the detector is simulated to be ≈0.09%. So, to reduce the overall amount of data transmitted, the frame data is zero-

[1]Number of hit pixels in a chip.

suppressed and only the addresses of the hit pixels are sent as the output. As shown in figure 2.2 taken from *ALPIDE operations manual* [5], two columns of pixels are read out as one double column. The figure also shows the order in which the hit pixels are read out from each double column, we call it the snake pattern for references in this Thesis. The readout of the matrix is organized as 32 regions of 16 double columns each. Each region is read out by 16 Priority Encoder circuits, each handling the readout of one double column. The data from the 32 region readout blocks are assembled and formatted by the Top Readout Unit module.

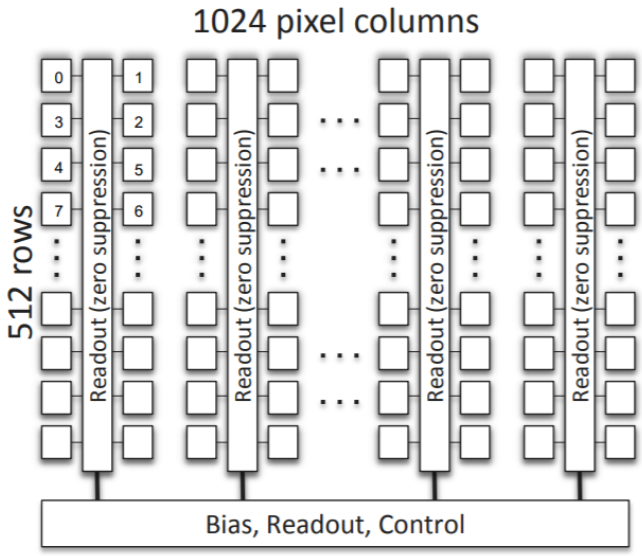

Figure 2.2: General architecture of the ALPIDE chip

The format of the data being sent out of the ALPIDE sensor is given in Table 2.1.

Data Word	Length (Bits)	Value (binary)
IDLE	8	1111_1111
CHIP HEADER	16	1010<chip_id[3:0]><BUNCH_COUNTER_FOR_FRAME[10:3]>
CHIP TRAILER	8	1011<readout ags[3:0]>
CHIP EMPTY FRAME	16	1110<chip_id[3:0]><BUNCH_COUNTER_FOR_FRAME[10:3]>
REGION HEADER	8	110<region_id[4:0]>
DATA SHORT	16	01<encoder_id[3:0]><addr[9:0]>
DATA LONG	24	00<encoder_id[3:0]><addr[9:0]>_0_<hit map[6:0]>
BUSY ON	8	1111_0001
BUSY OFF	8	1111_0000

Table 2.1: Data format adopted in ALPIDE sensor

2.2.2 ITS detector

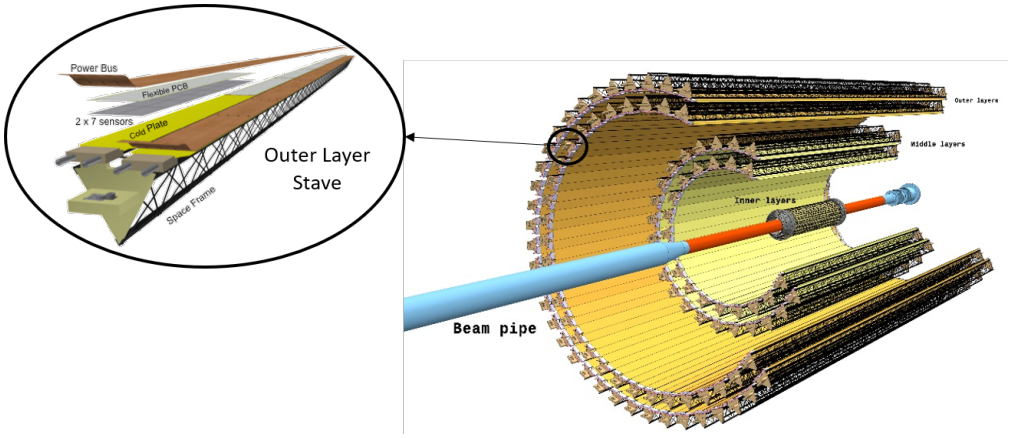

Figure 2.3: Layout of the new ITS detector

The Inner Tracking System consists of ≈24,000 ALPIDE sensors arranged in 7 cylindrical layers surrounding the collision point as depicted in figure 2.3 taken from [4]. The innermost 3 layers form the Inner Layer also known as the Inner Barrel. The next 2 layers form the Middle Layers and the

last 2 layers form the Outer Layers. The middle and outer layers constitute the Outer Barrel. Each layer consists of ALPIDE chips arranged as staves as shown in figure 2.3 taken from [5]. Each stave in the outer two layers is made up of 14 (2 x 7) outer barrel modules and each middle layer stave is made up of 8 (2 x 4) outer barrel modules, where an outer barrel module consists of 14 ALPIDE sensors as depicted in figure 2.4. Each inner layer stave/inner layer module consists of 9 ALPIDE sensors. Table 2.2 gives the exact count of the number of staves and sensors in each layer. With this complete arrangement, the ITS detector forms a 12.5G Pixel camera.

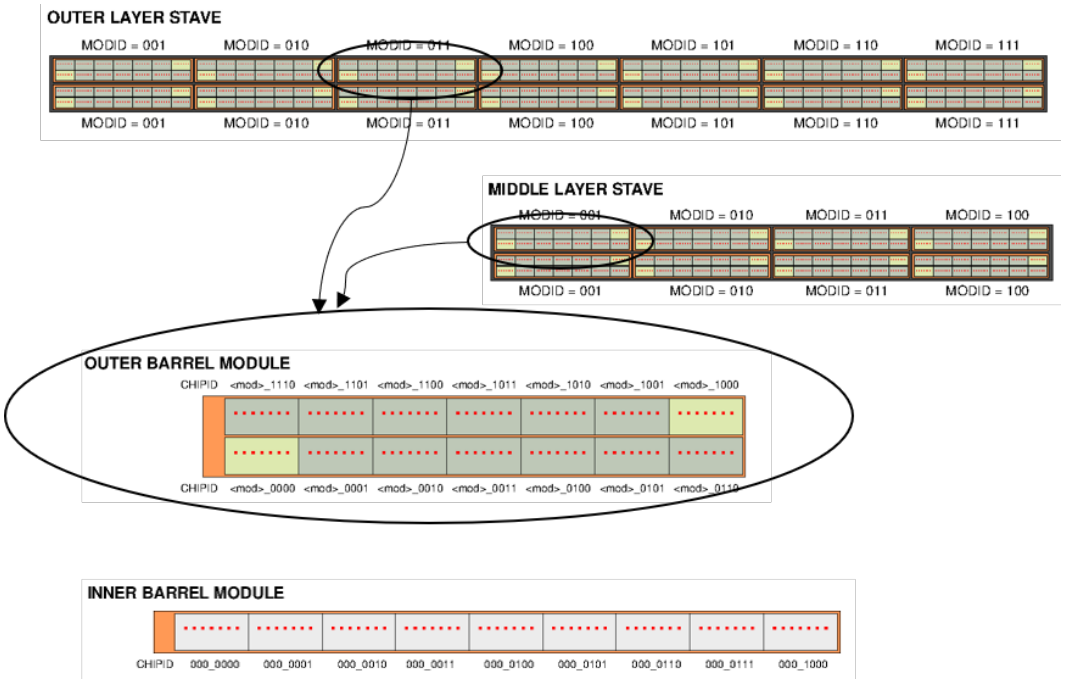

Figure 2.4: Arrangement of ALPIDE chips in Inner, Middle and Outer layer staves

Layer	No. of Staves	Sensors/stave	Sensors/layer	Pixels/layer
0	12	9	108	56.6M
1	16	9	144	75.5M
2	20	9	180	94M
3	24	8x14	2688	1.4G
4	30	8x14	3360	1.76G
5	42	14x14	8232	4.3G
6	48	14x14	9408	4.9G
Total	192		24120	12.5G

Table 2.2: Total pixels from ITS detector

2.3 Readout Electronics

Figure 2.5 shows the system that reads the data from the ITS detector and sends it to the Online-Offline (O^2) system located 100m away in a counting room where partial event reconstruction and event data reduction takes place before it is stored in the mass storage system.

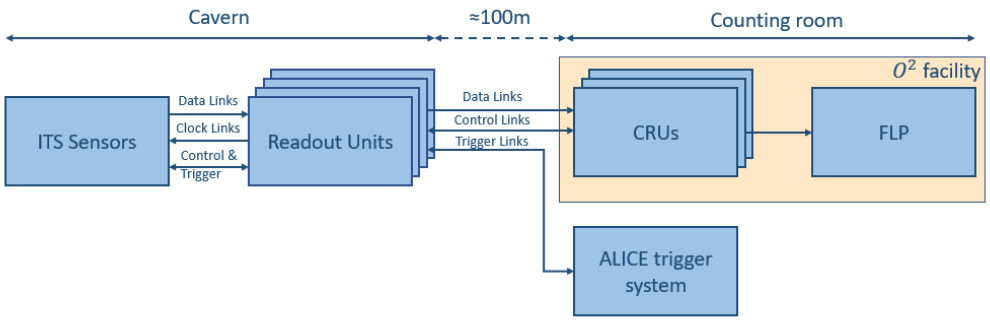

Figure 2.5: ITS readout electronics

A brief description of the blocks in the readout system is given below:

1. ALICE Trigger system: The ALICE trigger system generates the ap-

propriate triggers to the ITS as well as other detectors in ALICE. It consists of the Central Trigger Processor (CTP) and the Local Trigger Unit (LTU). The data from sensors can be read either continuously or in triggered mode. In triggered mode, for each collision, the CTP generates the trigger signal and the LTU modules distribute it to all detectors. This trigger starts the read-out of detectors.

2. Readout Units: The ITS readout system is composed of 192 Readout Units located 5m away from the outermost ITS layer. The readout units host FPGAs that configure and control the pixel sensors, receive and assemble data, and manage power units. Each readout unit is connected to one ITS stave. The data are read out via 3816 differential high-speed lines. Pixel sensors are controlled and monitored via 624 bidirectional buses. The GigaBit transceiver (GBTx) serializer/de-serializer present on the readout unit converts the electrical signals into optical signals to send the data for further processing and storage to the Common Readout Unit (CRU) over optical links.

3. Common Readout Unit (CRU): There are 24 CRU boards in the ITS readout system. It acts as an interface between the on-detector electronics, the First Level Processors (FLP), and the ALICE trigger system. It is based on high performance FPGA processors and is equipped with multi-gigabit optical inputs and outputs. The bi-directional front-end links based on the Versatile Link and GigaBit Transceiver (GBTx) serializer/deserializer ASIC connect the on-detector electronics to the CRU,

carrying detector data, configuration, and trigger information. The link bandwidth is 400 MB/s each.

4. First Level Processors (FLP): These are high-performance servers based on Intel Xeon CPUs with high speed 10Gb ethernet or fiber channel as its network interface. It receives a partial event from the CRUs attached to its PICe bus and sends it to the Event Processing Nodes (EPN) either directly or after local processing (e.g. cluster finding).

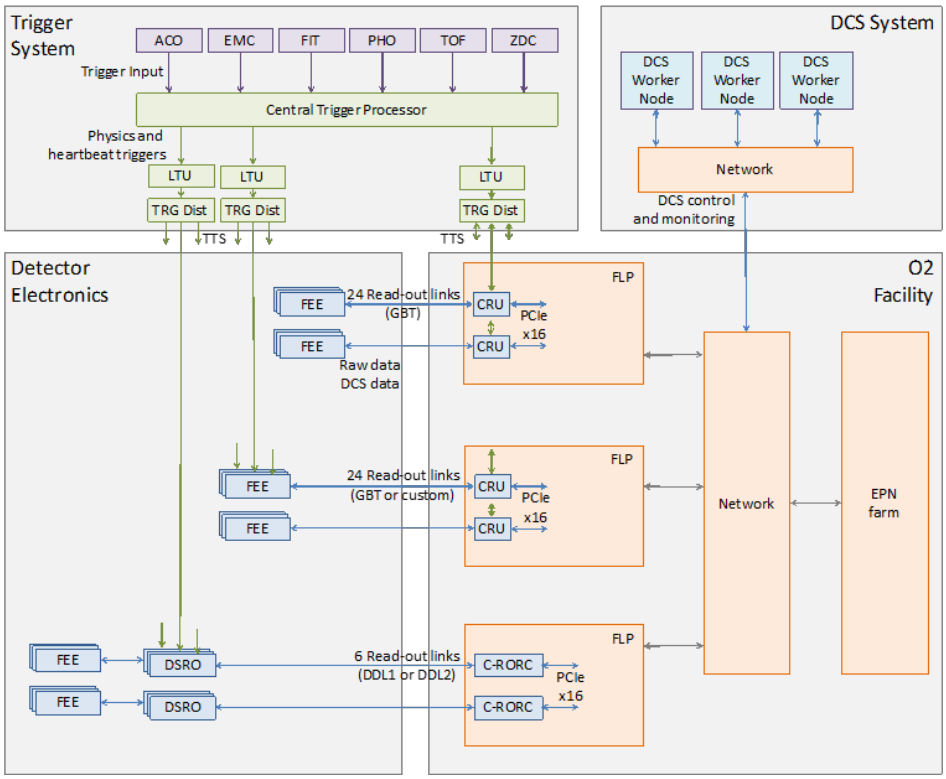

Figure 2.6: Global architecture of the Online and Offline Computing system (O^2)

The global architecture of the O^2 system is shown in figure 2.6. Each detector will send its data to the O^2 system. ITS first sends its data to the CRUs over GBT links which is then sent to the FLPs in the O^2 system over a PCIe interface. Before being stored to the mass storage system, the data will be compressed in 2 stages allowing more number of events to be recorded. The first level of compression is performed at a local level (e.g., cluster finding) in the FLP. Processing of global data from all sensors (such as reconstruction of tracks and association of them to primary vertex) in the EPNs allows for further reduction in data volumes [6].

2.4 Local compression - Cluster Finding

Instead of storing the raw data (i.e., pixel hit addresses) to the mass storage system, we can identify groups of adjacent pixels called clusters and store the relative position and shape information of clusters instead. Most of the cluster shapes repeat, giving rise to redundant information being transmitted and stored. Huffman compression [7] can be used to exploit this redundancy and compress the data.

If the data is compressed even before sending the data to the FLPs, the computing nodes needed in the FLPs can be significantly reduced. The resources on the CRU are not completely utilized, so these are the potential resources that could be utilized for Cluster finding and Huffman compression. A study was performed to identify the best place to perform these operations. Figure 2.7 shows the different scenarios that were considered. If the compres-

sion is performed before the EPN, it needs to be decompressed again in EPN as the data is needed for global event reconstruction. Though the bandwidth needed for transmitting compressed data is low, the amount of processing needed to decompress the data overshadows this benefit. Therefore, the compression will be performed in the EPN and not in the stages before. The only 2 options (marked with green tick in figure 2.7) to evaluate are:

1. Performing clusterization on the CRU
2. Performing clusterization on the FLP

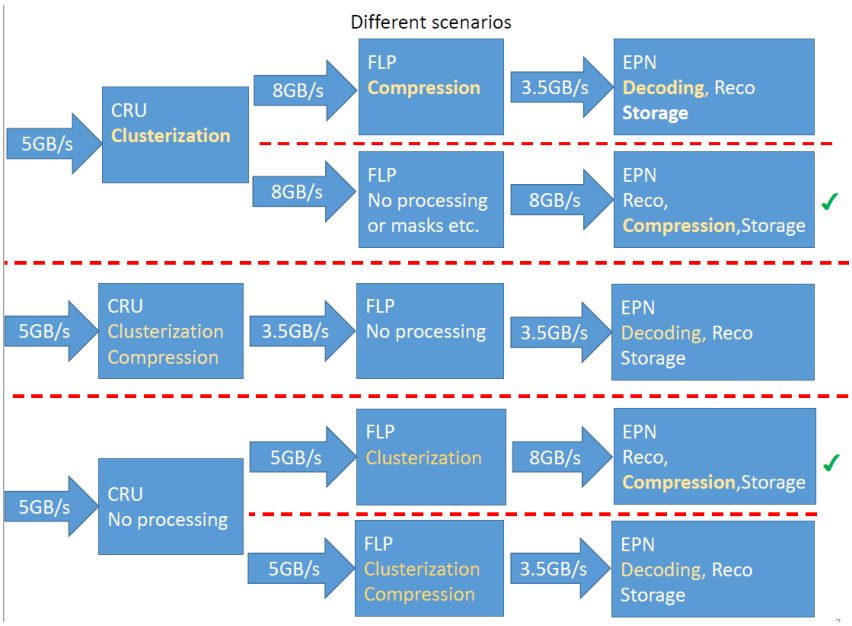

Figure 2.7: Scenarios considered for local compression of ITS data

In this Thesis, we are evaluating the first option to find out if the

Cluster Finder algorithm can be implemented with the resources available on the CRU.

Chapter 3

Cluster Finding

This chapter dives deeper into cluster finding and discusses related work. It is divided into three sections. The first section gives some definitions and briefly explains a generic 'Connected Component Labeling' algorithm [8]. The second section discusses some existing work in Cluster Finding with the focus on hardware implementations. The third section introduces prior work done at CERN and the algorithm that was developed in software, intended to be used in the FLP.

3.1 Some definitions and generic algorithm

3.1.1 4-connected component

"A pixel, Q, is a 4-neighbor of a given pixel, P, if Q and P share an edge" [9]. As shown in figure 3.1, pixels P2, P4, P6 and P8 are the 4-neighbors of pixel P. A connected component is called a 4-connected component if every 2 pixels that are adjacent are 4-neighbors.

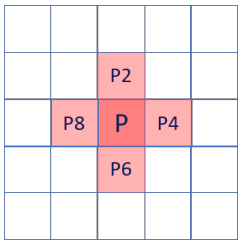

Figure 3.1: 4-connectivity

3.1.2 8-connected component

"A pixel, Q, is an 8-neighbor of a given pixel, P, if Q and P either share an edge or a vertex" [9]. As shown in figure 3.2, pixels P1, P2, P3, P4, P5 P6, P7 and P8 are the 8-neighbors of pixel P. A connected component is called a 8-connected component if every 2 pixels that are adjacent are 8-neighbors.

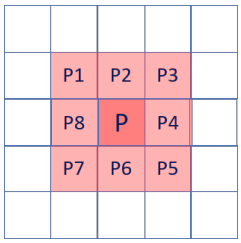

Figure 3.2: 8-connectivity

3.1.3 Generic two-scan algorithm for Connected Component Labeling

Figure 3.3 taken from [8] shows the generic 2-scan algorithm for connected component labeling of a 4-connected component.

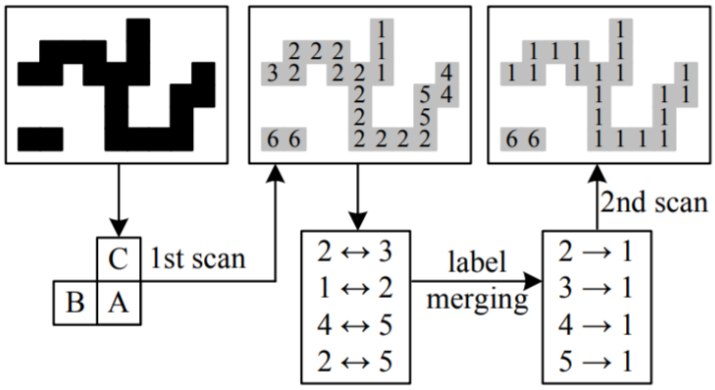

Figure 3.3: Two-scan labeling algorithm

It shows the 3 phases of binary image (I) involved. The black/gray pixels are the object pixels whereas the white pixels are the background pixels. During the first scan, a labeling window scans the entire binary image in raster scan order and assigns initial labels to all the pixels. A label is generated for pixel A (i.e. I(x, y)) of the labeling window. Pixels B (i.e. I(x-1, y)) and C (i.e. I(x, y-1)) are the neighboring pixels. For the current pixel I(x, y), it does nothing if it is a background pixel. If it is an object pixel, it processes it as follows:

- If its left neighbor I(x-1, y) and upper neighbor I(x, y-1) are background pixels, a new label is assigned to I(x, y).
- If both left neighbor I(x-1, y) and upper neighbor I(x, y-1) are labeled already, the label of left neighbor I(x-1, y) is copied and the labels of I(x-1, y) and I(x, y-1) are marked as equivalent.

- If left neighbor I(x-1, y) is labeled already, the label of I(x-1, y) is copied to I(x,y).
- If upper neighbor I(x, y-1) is labeled already, the label of I(x, y-1) is copied to I(x,y).

The pseudo code for the above algorithm can be given as follows:

Listing 3.1: Pseudo code for CCL

```
1 l = 0
2 for each I(x,y) in raster scan
3 |if I(x,y)==0 // I(x,y) is a background pixel
4 || label[I(x,y)] = 0
5 |else  // I(x,y) is an object pixel
6 ||if I(x-1,y) == 0 and I(x,y-1)==0
      //Both neighbors are background pixels
7 ||| l++
8 ||| label[I(x,y)] = l
9 ||else if I(x-1,y) == 1 and I(x,y-1)==1
      //Both neighbors are object pixels
10 ||| label[I(x,y)]=label[I(x-1,y)]
      // copy label of I(x-1,y)
11 ||| mark labels of I(x-1,y) and I(x,y-1) as
      equivalent
12 ||else if  I(x-1,y) == 1
13 ||| label[I(x,y)]=label[I(x-1,y)]
      // copy label of I(x-1,y)
14 ||else // I(x,y-1) == 1
15 ||| label[I(x,y)]=label[I(x,y-1)]
      // copy label of I(x,y-1)
16 ||end_if
17 |end_if
18 end_for
```

During the second scan, the equivalent labels are merged, and all the image pixels are re-labeled with merged labels. This concludes the connected component labeling.

3.2 Hardware implementations of Connected Component Labeling

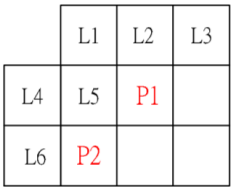

Figure 3.4: 3x4 label assigning window

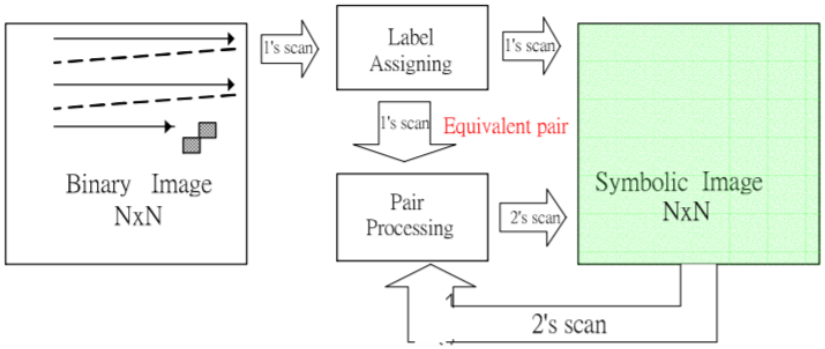

Figure 3.5: Connected Component Labeling flow

In [10], Yang et al. have proposed an efficient architecture for fast label assignment in binary images. In their work, they use a 3x4 window of incoming pixels (as shown in figure 3.4) which allows them to concurrently process two label assigning operations (for P1 and P2). Figure 3.5 shows the connected component labeling flow adopted in this work. The label assigning block assigns initial labels to the pixels as well as finds label pairs in the connected components. To resolve the label mergers, the authors in [10] have

proposed the use of a class register array. The symbolic image after the first scan is stored in memory and a second scan is needed to resolve the component mergers. This falls under the category of two-pass algorithms as it needs two raster scans of the image to obtain the final connected component. The drawback of this implementation is the memory required to store the complete image as well as the latency.

The authors in [8] propose an implementation to accelerate both label generation and merging of equivalent labels. They perform label generation for four pixels in parallel by using a 2x5 window. The first scan is used to generate temporary labels to establish the label equivalences, while the actual labels are not written to memory, thus requiring less memory compared to other two-pass algorithm implementations. A special lookup table, as shown in figure 3.6, is implemented to merge the equivalent labels. For each label, an assignment and successor labels are stored. With each equivalence, the assignment and the successor are accessed successively until the merger chains have been resolved. This label merging takes several clock cycles depending on the length of the merger chain.

label (address)	assignment	successor
1	-	4
2	1	3
3	1	-
4	1	5
5	1	2

Figure 3.6: Label Merging Lookup Table

Two-pass implementations have a latency of 2 image frames and need sufficient memory to store at least one complete image. If on-chip memory is not sufficient, this requires the use of external high bandwidth memory. To overcome these two issues, the authors in [11], [12], [13] use single pass analysis and gather data on the regions as they are built, avoiding the need to store intermediate data. The work in [11] proposed merging and relabeling on-the-fly. This was achieved by maintaining a merger table. A merger control block updates the merger table whenever two objects are merged. Merger chains were resolved by maintaining a chain stack data structure. To enhance the performance of this approach, the authors in [12] streamlined the architecture to operate on parts of the image in parallel and then coalesce these parts at the end. The paper proposed the use of local labels and global labels to avoid errors due to the same label being applied to different parts of the image when operating on them in parallel. In [13], the authors attempted to make the design resource efficient. They reduced the memory requirements by re-using the labels that are no longer needed. To support this, they implemented out-of-order processing of labels.

3.3 Prior work - Software implementation

Work to assess the performance of different hardware and programming models for cluster finding is going on in parallel by different groups working on ALICE. Besides implementing the Cluster finder on the CRU, the other option being considered is to implement it in software on the FLP. Here we

present the algorithm that was written in C++ by Chapeland, S et al. [3], initially intended to be deployed on the FLP. We use this work as the starting point for our FPGA implementation. In this Thesis, we have implemented all the features supported by this piece of software.

3.3.1 Background

As discussed earlier in section 2.4, the aim is to achieve compression by first identifying clusters of pixels and later using Huffman compression on the cluster information. Figure 3.7 depicts the overall idea.

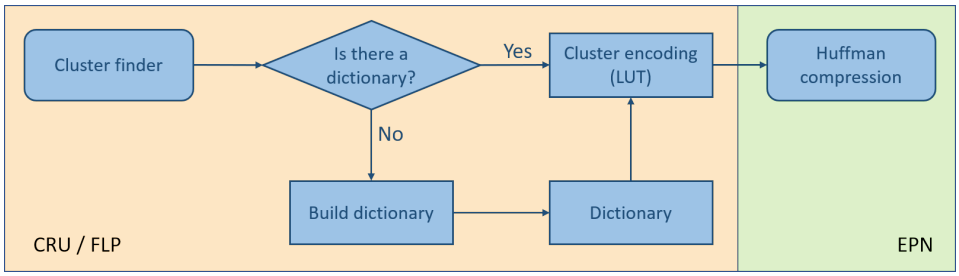

Figure 3.7: An overview of ITS cluster finder

From the incoming pixel hit data from ALPIDE, clusters will be identified. A dictionary data structure will be maintained to record already observed cluster shapes. The dictionary will be arranged in descending order of the number of occurrence of the cluster shapes, thus assigning a lower id to the most frequent shape. This information will be useful for Huffman compression. Every time a cluster is identified, its shape will be checked against already identified cluster shapes in the dictionary. If it does not exist, it will

be added to the dictionary. If it is present, its Id will be looked up in the dictionary and this Id will be sent to the EPN for Huffman compression.

3.3.2 C++ Implementation

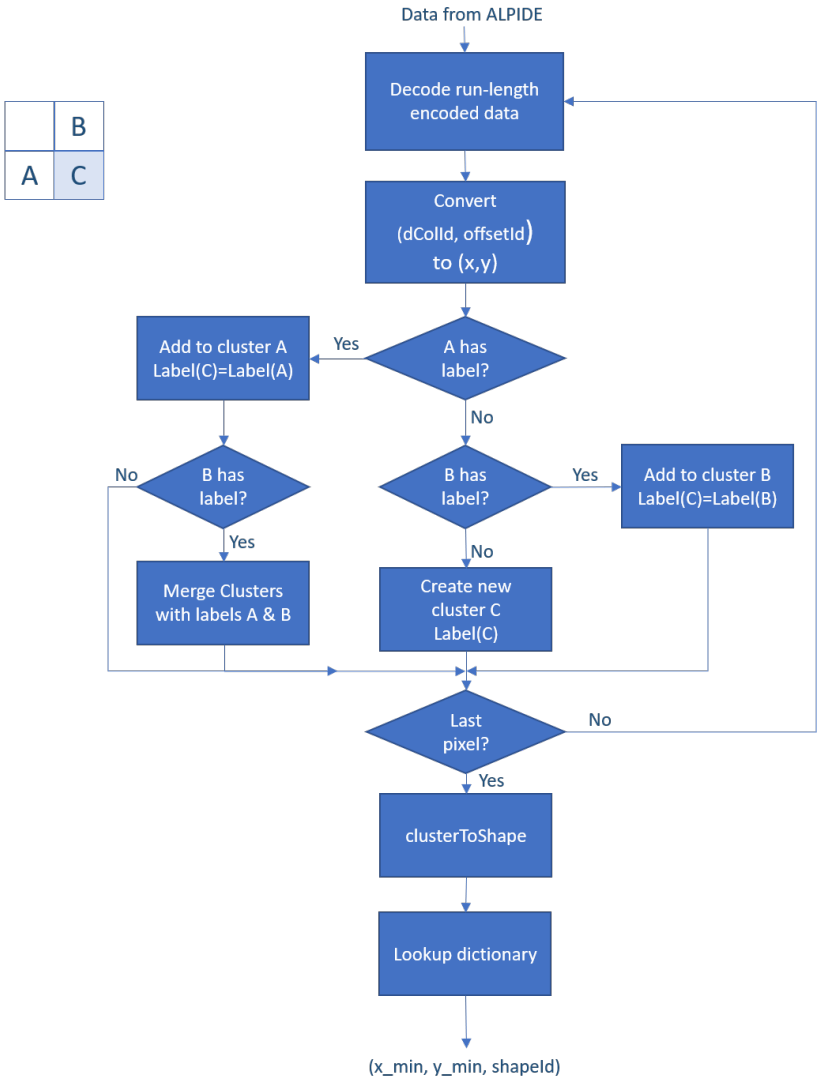

Figure 3.8: Software implementation flowchart of Cluster Finder

We receive the data from ALPIDE in a snake pattern as discussed previously in section 2.2.1. With this algorithm, we are assigning a label to pixel C. Pixel A & B should already be processed due to the order in which input pixels arrive. The incoming data is processed as follows:

- Decode the run-length encoded input. Extract hit pixel information from the DATA_SHORT[1] or DATA_LONG[2] data words. The pixel address is encoded as double column ID and offset within the double column.
- Convert (dColId, offsetId) to (x,y).

```
int y=(offsetId)/2;
int x=dColId*2 + (offsetId & 0x1) ^ ((offsetId & 0x2) >> 1);
```

- Check adjacent labels of A[3] & B.
  - If only A is labeled, copy its label to C and add pixel C to A's cluster.
  - If only B is labeled, copy its label to C and add pixel C to B's cluster.
  - If both A & B are labeled, copy A's label to C, merge clusters A & B and add C to that cluster. The merging operation consists of

[1] 1 pixel.
[2] 1 hit pixel, followed by next 7 pixels in the double column.
[3] A can either be in the same double column or previous double column, in which case, a buffer containing labels of previous double column needs to be maintained.

copying all the pixels from cluster B to cluster A and freeing cluster B.

  - If neither of A & B are labeled, create a new cluster with C.

- Repeat the above process until the CHIP_TRAILER data word is found, which marks the end of the chip data.

- Now, for all clusters, find the shapeBits. This operation involves finding the minimum x and y and assigning 1 or 0 to bits in an 8x8 window as seen in the code snippet below

```
for(auto hit: cluster.hits) {
  if (hit.X<xmin) xmin=hit.X;
  if (hit.X>xmax) xmax=hit.X;
  if (hit.Y<ymin) ymin=hit.Y;
  if (hit.Y>ymax) ymax=hit.Y;
}

// bit-encode shape
shape.shapeBits=0;
for(auto hit: cluster.hits) {
  int x=hit.X-xmin;
  int y=hit.Y-ymin;
  shape.shapeBits[x+y*ClusterShape::shape1Dsize]=1;
}
```

- The code works in two modes:

  1. buildDictionary mode

     - If dictionary entry for shapeBits exists, increase shape occurrence count.

     - If entry does not exist, add a dictionary entry and set shape occurrence count as 1.

2. useDictionary mode
   - If dictionary entry exists, lookup index of that shape and send to output (x_min, y_min, shapeID)
   - If entry does not exist, send (x,y) of the pixel.

This algorithm takes 47.59 ms to process the data from 2124 sensors as measured on an Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz, 4 cores, single socket, 8GB RAM machine. The performance will be better on the FLPs, which will have advanced server grade processing capabilities. Data from all the ALPIDE sensors is processed sequentially, one after the other. Thus, for processing data from one sensor it takes $\approx$22.4μs. In the actual system, we will be receiving data at an average rate of 1 event/20μs. The processing time required by the software implementation implies a significant pile-up rate. Thus, we need a system that can readout and process the data fast. This is another important motivation to implement the system in hardware, where we can make use of pipelined and parallel processing to speed up the process.

Thus, we establish the below goals for the study done in this Thesis.

- Estimate the resources needed for implementing this algorithm on the CRU FPGA (Altera Arria10)
- Estimate the time needed to process the data on the FPGA (We have 24 CRUs in the system and within these CRUs, we can have multiple instances of the 'Cluster Finder'. So, we can process data from multiple sensors in parallel).

- By implementing this on the CRU, reduce the computing nodes needed on FLP.

Chapter 4

Cluster Finder - RTL Development

4.1 System Requirements

4.1.1 Sample input

The specification of the sensor inputs important to us are given in table 4.1.

Interaction rate (Pb-Pb)	50 kHz
Shaping time	5us
Acquisition window	20us
Noise	$10^{-6}\ px^{-1}$

Table 4.1: Sensor specifications

The corresponding data was generated from Monte Carlo simulations and converted to ALPIDE raw data format using the Aliroot[1] framework [14]. Figure 4.1 shows one such event for a single ALPIDE chip. As observed, the image is mostly empty with some groups of pixels (clusters) scattered throughout.

[1] ALICE Off-line framework for simulation, reconstruction and analysis.

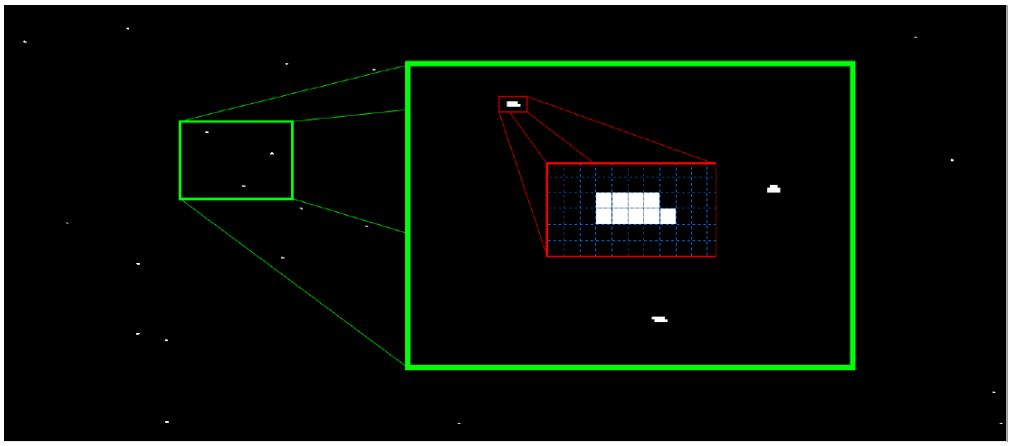

Figure 4.1: Image for one ALPIDE chip (zoomed)

A study was performed to estimate the maximum number of fired pixels per chip in each layer. The results of this study for the specifications described above are presented in table 4.2.

Layer	Number of fired pixels/chip
0	389
1	217.46
2	141.37
3	4.54
4	2.98
5	1.75
6	1.47

Table 4.2: Maximum Occupancy

From this study, it is evident that the number of fired pixels per chip is higher in the Inner Layer and reduces drastically for Middle and Outer Layers. We observe that the maximum number of fired pixels/chip is 389. Assuming a ≈20% margin, we can consider this number to be approximately 500 for the purposes of our system implementation.

4.1.2 CRU and System Integration

The CRU will be implemented on Intel's Arria 10 (10AX115S2F45I1SG) FPGA; its resources are summarized in table 4.3. As shown in figure 4.2, the input to the CRU comes from the APLIDE sensors and the Readout Units over 8 lanes. The data from multiple ALPIDE sensors are multiplexed on each lane (for the evaluation done in this Thesis, we assume that we receive the data from one ALPIDE sensor at a time followed by the next; not multiplexed). On the output side, it is connected to the FLP via a PCIe interface.

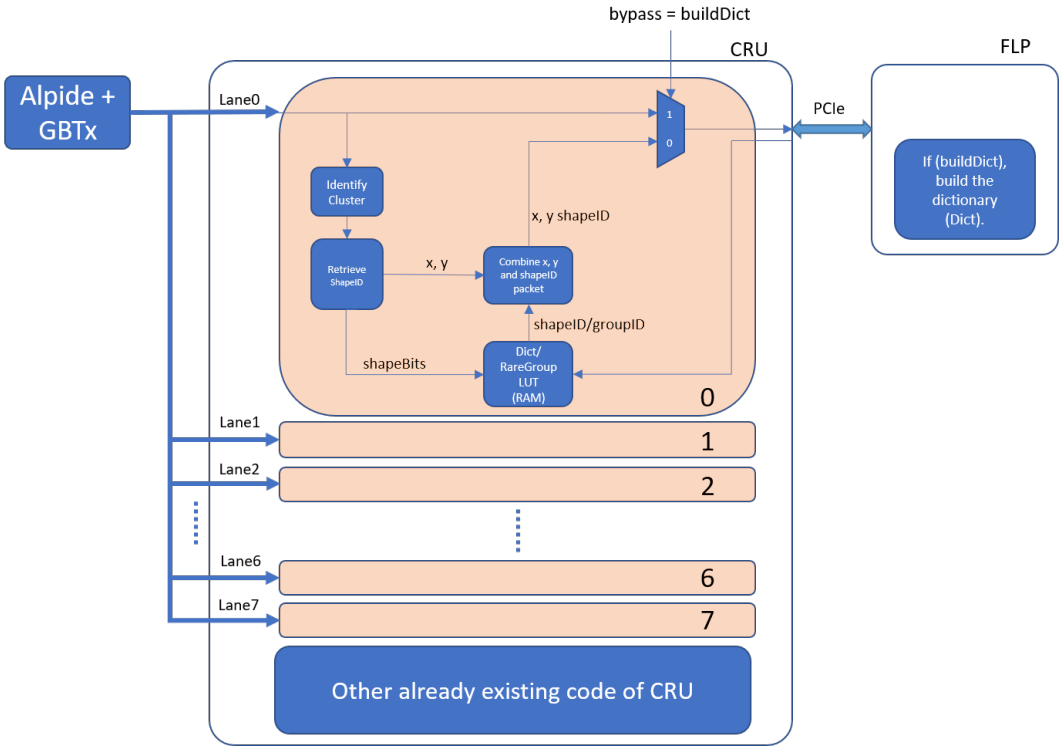

Figure 4.2: Integration of Cluster Finder with the ITS Readout System

In the existing CRU firmware implementation, there is an empty block called 'User Logic' where we will be integrating our Cluster Finder. The CRU operates with a clock frequency of 240MHz. We will be using this clock as an input to our block.

We envision the complete system to operate as described below.

- In the beginning, the Cluster Finder will operate in buildDict mode where the user logic, shown in instance 0 of figure 4.2, will be bypassed. The sensor data will be used as conditioning data by the FLP to build the dictionary database. This dictionary will then be sent to the Cluster Finder over the PCIe interface.
- Once the dictionary is constructed and uploaded to the FPGA, we will look up the shape in the dictionary to find the ShapeId for the incoming sensor data from that point onwards.
  - First, identify the cluster
  - Obtain corresponding shape bits
  - Lookup dictionary and get ShapeId
  - Send (x_min, y_min, ShapeId) over the PCIe interface

There will be multiple instances of this Cluster Finder that will operate on the incoming data from 8 lanes in parallel as depicted in figure 4.2.

As the CRU testbench is not available, for our development purposes we will be feeding the simulated sensor data from the PCIe interface as well.

These data will be received by the PCIe hard IP and sent to the Cluster Finder over an AvalonMM [15] interface.

4.1.3 Available resources

Table 4.3 shows the available Arria 10's resources for our purposes:

Resource	**Total**	**Used by CRU logic**	**Available for Cluster Finder**
ALM (Adaptive Logic Module)	427,200	74,915 (18%)	352,285 (82%)
Block Memory bits	55.562 Mbits	8.672 Mbits (16%)	46.89 Mbits (84%)
DSP blocks	1,518	0 (0%)	1,518 (100%)

Table 4.3: Available Arria 10 resources

4.1.4 Specifications Summary

Clock Rate	240 MHz
Available processing time	20 us
Available ALMs	352,285
Available Block Memory bits	46.89 Mbits
Image size	1024x512 pixels

Table 4.4: Specifications Summary

4.2 Top-level architecture

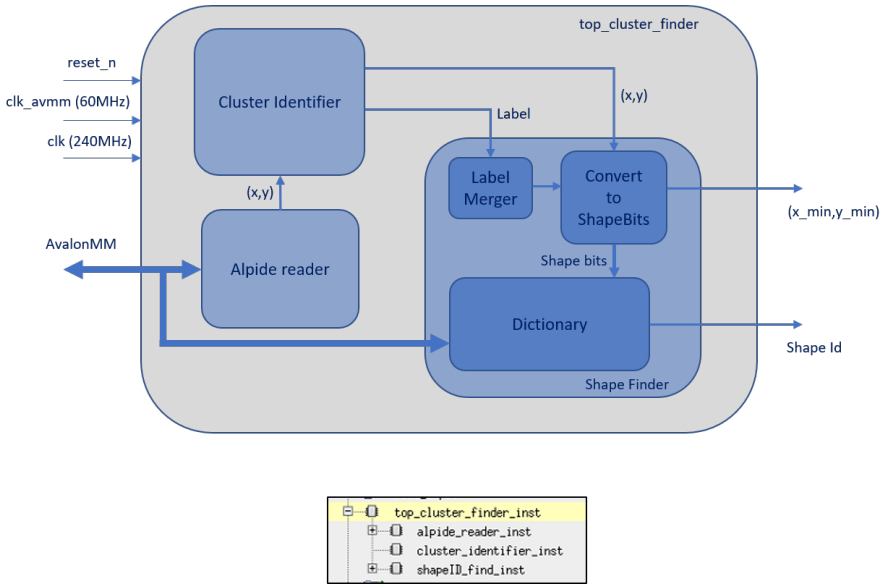

Figure 4.3: Cluster Finder - Top-level Architecture

Dictionary data and run-length encoded ALPIDE data are sent over the AvalonMM interface running at a clock frequency of 60MHz. The rest of the logic runs at a clock frequency of 240MHz. The dictionary data consisting of 1000 entries of 64bits each are written to the dictionary memory (1000 x 64bits). The ALPIDE data are decoded in the alpide_reader block. The decoded (x,y) coordinates are fed to the cluster_identifier block where each pixel is labeled based on the cluster to which they belong. This is the initial labeling; component mergers are taken care of in the next block. The shapeID_find block merges the components, finds x_min, y_min & shapeBits (8x8 block) for the clusters, finds the shapeID from the dictionary and sends it to the output.

4.3 Block-wise Implementation and Evaluation

4.3.1 ALPIDE reader

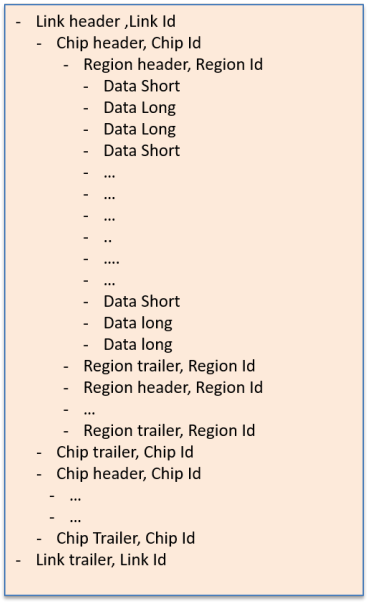

Figure 4.4: Sensor Data Encoding

Figure 4.4 shows the format in which ALPIDE data are encoded. The data from all the chips belonging to a link are enclosed within Link Header and Link Trailer along with the corresponding 'linkId'. The data of all the 32 regions within a chip are enclosed between Chip Header and Chip Trailer along with the corresponding 'chipId'. Similarly, DATA_SHORT & DATA_LONG belonging to a region are sent after the corresponding Region Header and 'regionId'. As was seen in table 2.1, the length of these data words is not constant, they are variable. Data words are either 8, 16, or 24 bits. As the smallest granularity is 8 bits, it was chosen as the unit for processing incoming

data. The FSM (Finite State Machine) to decode these data is shown in figure 4.5. Each state in the FSM operates on 8-bit input data. If a data word is more than 8 bits long, the FSM uses multiple states (colored with the same color).

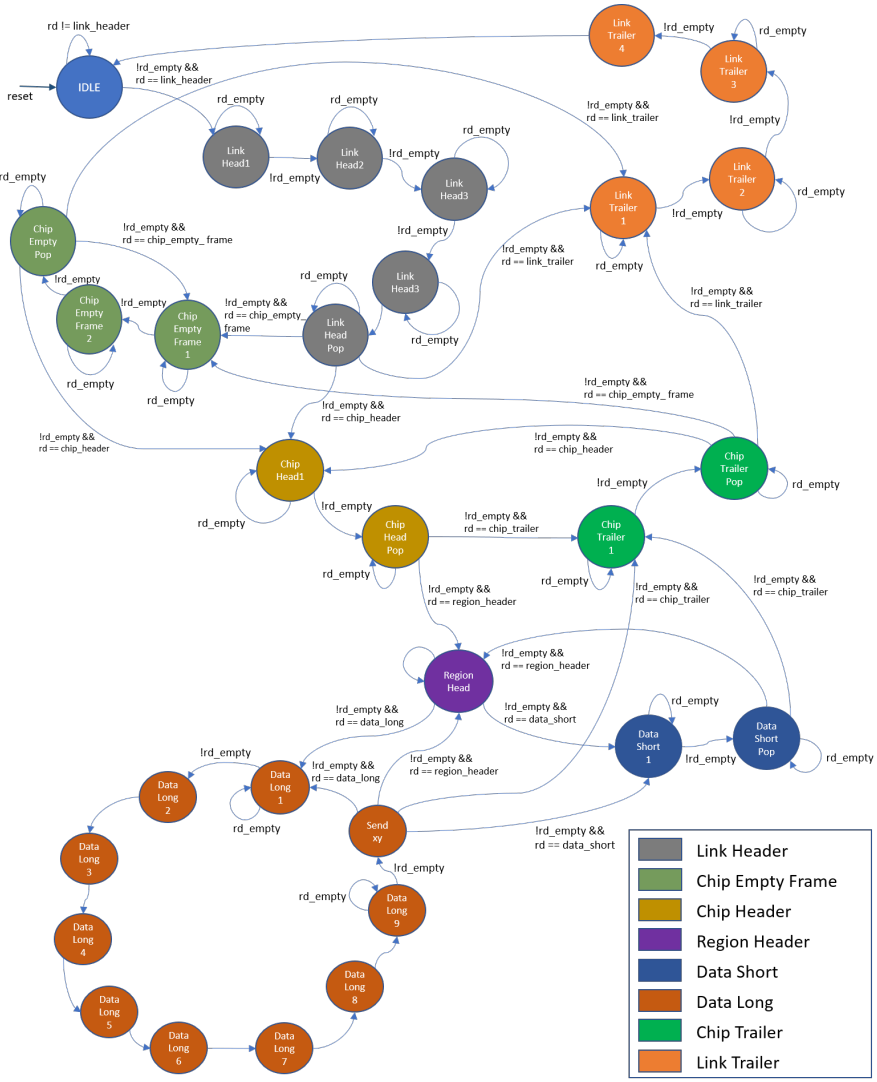

Figure 4.5: Decoder FSM

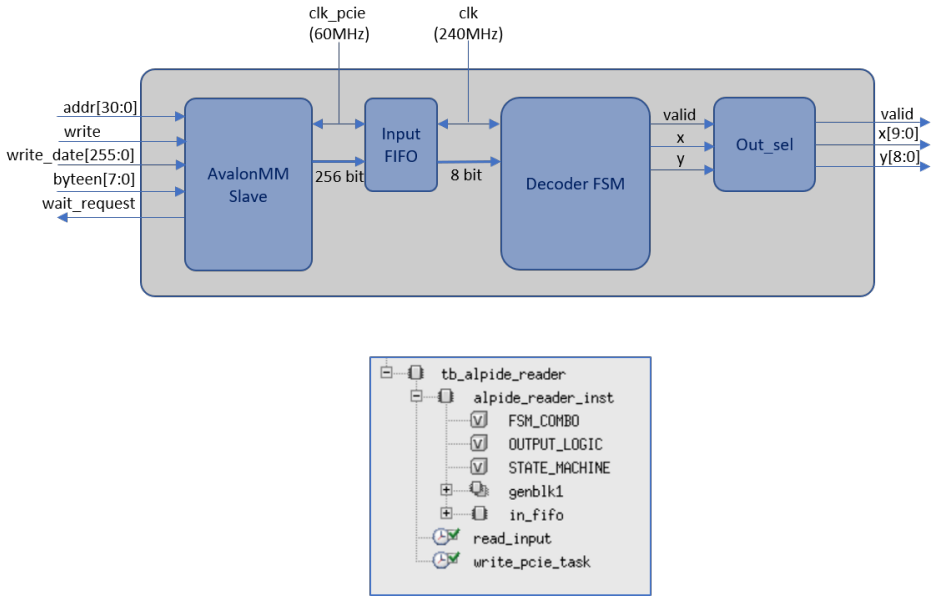

Figure 4.6: Alpide Reader block

Figure 4.6 shows the block diagram for the 'alpide_reader'. The avalonMM data is 256 bits wide, whereas the FSM reads only 8 bits at a time. Also, the frequency of the input is not same as the rest of the logic. Therefore, in order to decouple the input interface and the FSM, we added a dual clock mixed width FIFO of size 512 bytes (256bits x 16) with 256bit wide input and 8 bit wide output. The wait states on the AvalonMM interface take care of the FIFO full condition. The FIFO size can be increased if the block feeding the 'alpide_reader' cannot tolerate wait states. The FSM reads 1 byte at a time from the FIFO and produces as output the x (10 bits), y (9 bits) and a valid signal corresponding to each hit pixel decoded from DATA_SHORT and DATA_LONG. The out_sel logic gates the output based on the 'chipId' and

'linkId' configured. The output waveforms of this block are shown in figure 4.7.

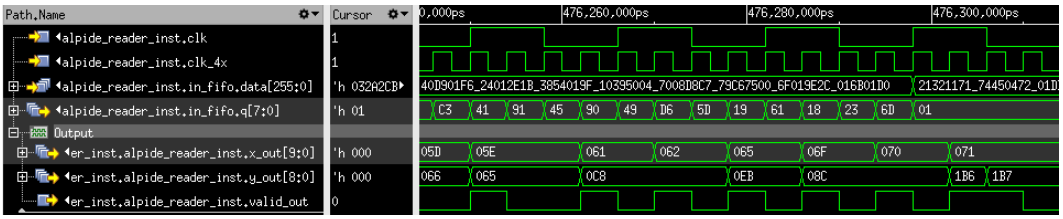

Figure 4.7: Alpide Reader waveforms

4.3.2 Cluster Identifier

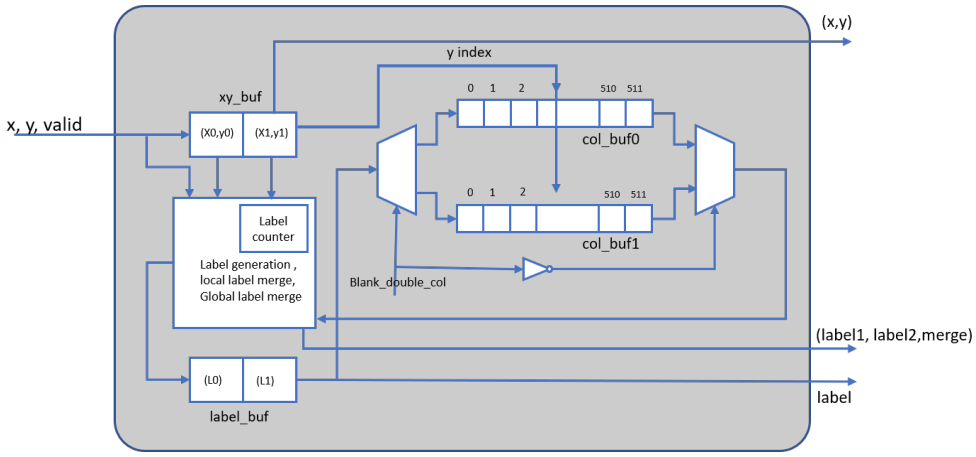

Figure 4.8: Cluster Identifier block

This block is responsible for applying labels to the identified clusters as well as local level component merging. Larger mergers such as combining two existing clusters are deferred to the next block (see section 4.3.3). It identifies labels that need to be merged and sends an indication to the next block. In the

software implementation, clusters are combined by copying over one cluster to the other whenever a label equivalence is found. This is a time consuming task. We take advantage of the parallelism of hardware to avoid this iterative process. This is taken care of in the next block and will be discussed in section 4.3.3.2.

4.3.2.1 Label assignment and mergers

As seen in section 4.1.1, there will be at most 500 pixel hits per sensor. In the worst case, if no two pixels are neighbors, there will be maximum of 500 clusters having 1 pixel each. Thus, our system should be able to generate 500 distinct labels. We generate these labels with a 9-bit counter (label_cnt[8:0]) which generates labels from 1 to 511.

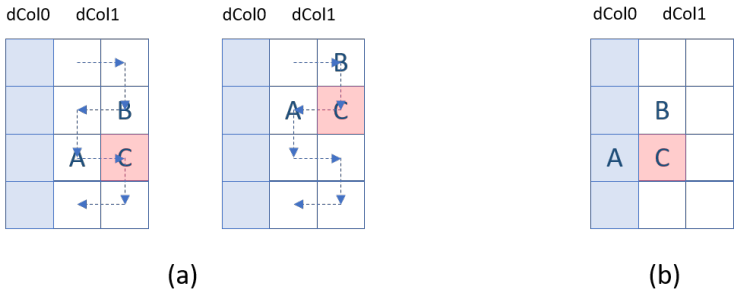

Figure 4.9: Adjacent pixels – Different cases

As discussed in section 3.3.2, while determining the label of a new pixel C, its adjacent pixels A & B should be checked for existing labels. Since the pixel data are sent for each double column and not single columns, one of the

two cases, as depicted in figure 4.9, can occur.

- Pixel A is in the same double column (figure 4.9(a)): For this case, it can either be the immediate previous pixel or the immediate future pixel (due to the pixel data arriving in the snake pattern). For this case to be checked, we need to save the previous two pixels. We save this information in buffers 'xy_buf' and 'label_buf' of sizes (10+9)x2 and 9x2, i.e., a total of 76 bits.

- Pixel A was in the previous double column (figure 4.9(b)): For this case to be checked, we need to buffer the labels of the second column of the previous double column. We have used internal buffers 'col_buf0' & 'col_buf1' of size 512 (no. of pixels in a column) of 9 bits each (i.e., 2x576 bytes) for this purpose. One of them acts as the previous double column and the other one is used to store labels of the current double column. When we proceed to the next double column, we swap the buffers.

4.3.2.2 Component mergers

Since the input pixels arrive in a snake pattern, several cases arise that need to be considered while identifying the labels to be merged. These cases are summarized in figure 4.10. The identified labels (pix_label1, pix_label2) are sent to the output along with a 'merge' signal.

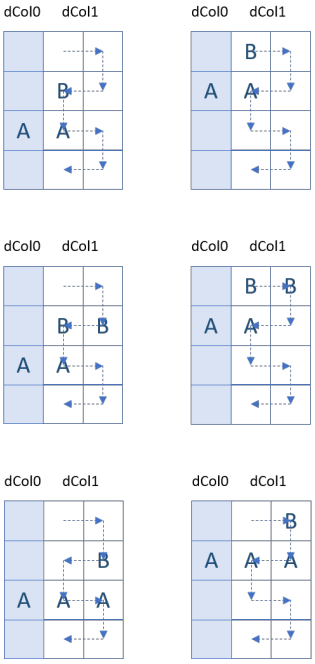

Figure 4.10: Component Merger - Scenarios

The waveforms for input and output signals of this block can be seen in figure 4.11. As can be seen from the waveforms, the latency of this block is 2 clock cycles (8 ns).

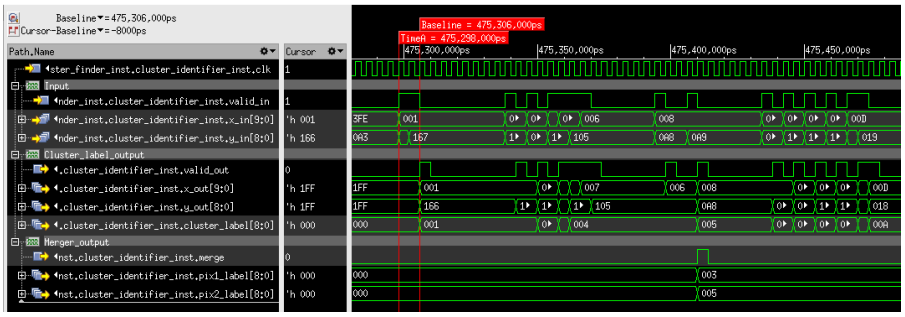

Figure 4.11: Cluster Identifier waveforms

4.3.3 ShapeId finder

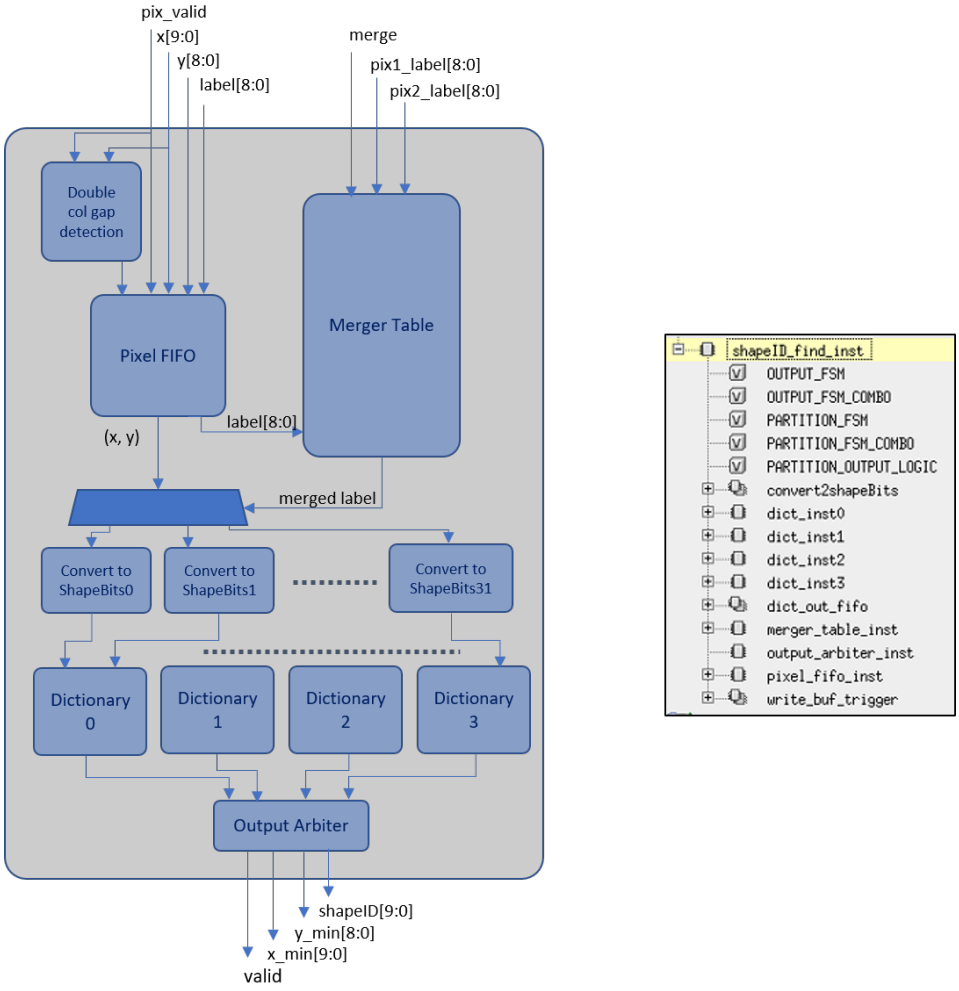

Figure 4.12: ShapeId Finder

The 'shapeId_find' block is responsible for merging the clusters with equivalent labels, finding shapeBits for merged clusters and looking up the dictionary and sending the corresponding 'shapeId'.

The merger table records the equivalent labels and resolves the merger chains. The pixels belonging to different clusters are partitioned and sent to respective 'convert2shapeBits' blocks by the partitioning controller. The 32 'convert2shapeBits' blocks read pixels belonging to each cluster and find the shapeBits corresponding to an '8x8' block. The dictionary block contains the dictionary sent by the FLP and is used to look up the 'shapeId' corresponding to 'shapeBits'. Since we are processing 32 blocks in parallel, there has to be an arbiter at the output. This is a round robin arbiter; it selects the output from each dictionary one after the other and presents the 'shapeId' at the final output. In the sections below we discuss each sub-block in detail.

4.3.3.1 Merger table

The merger table is a dual port RAM[16] that stores equivalent labels. It has 512 entries of 9 bits (i.e. a total of 576 bytes) each, corresponding to each label. One port is used for writing the equivalent label and the other port is used for reading the label. Figure 4.13 shows a sample memory content for 8 labels for the shown equivalences.

Label (address)	Equivalent label (data)
1	1
2	2
3	1
4	4
5	2
6	6
7	2
8	1

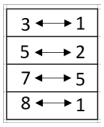

Figure 4.13: Merger Table and Label Equivalences

The memory will be initialized such that each memory location contains its own address as its data, i.e., each label will point to itself (for example, address 0x1 will hold 0x1 as its data, address 0x2 will hold 0x2 as its data, and so on). Every time equivalent labels are found, the value of the smaller label is written to the address pointed to by the larger label. For example, for the equivalence between 3 & 1, the value of 1 is written to address 3. Chain mergers are also taken care of while writing the equivalent label. For example, in figure 4.13, labels 5 & 2 are equivalent and labels 7 & 5 are equivalent. Initially, the equivalence of 5 & 2 is resolved by writing 2 to address 5. Then, while resolving the equivalence of 7 & 5, first the content of location 5 is checked. Since this contains 2, the content of 2 is checked. As its content is 2, this is the smallest label in the merger chain. Thus, this value, 2, is written to address 7. So, 7 and 5 both now point to 2. Any further chain, for example 12 equivalent to 7, will be resolved after just 2 memory reads since 7 already points to 2. So, now, 12 will also be pointing to 2.

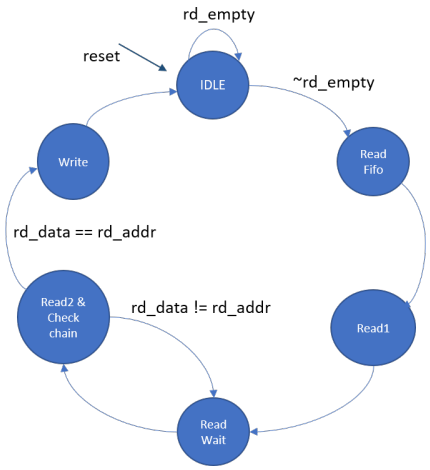

Figure 4.14: Merger Control FSM

The merger control FSM shown in figure 4.14 controls the process described above. Figure 4.15 shows the block diagram of the 'merger_table' block. The equivalent labels from the input are stored in the 'Merger FIFO'. This gives the FSM enough time to resolve the chained equivalences. The merger control FSM reads data from address 'small_label' over port0. It continues to read from memory until the data matches its address. This marks the end of the chain. This value is now written to the address 'big_label'. Now, the equivalent label after the chain resolution can be read from the other port.

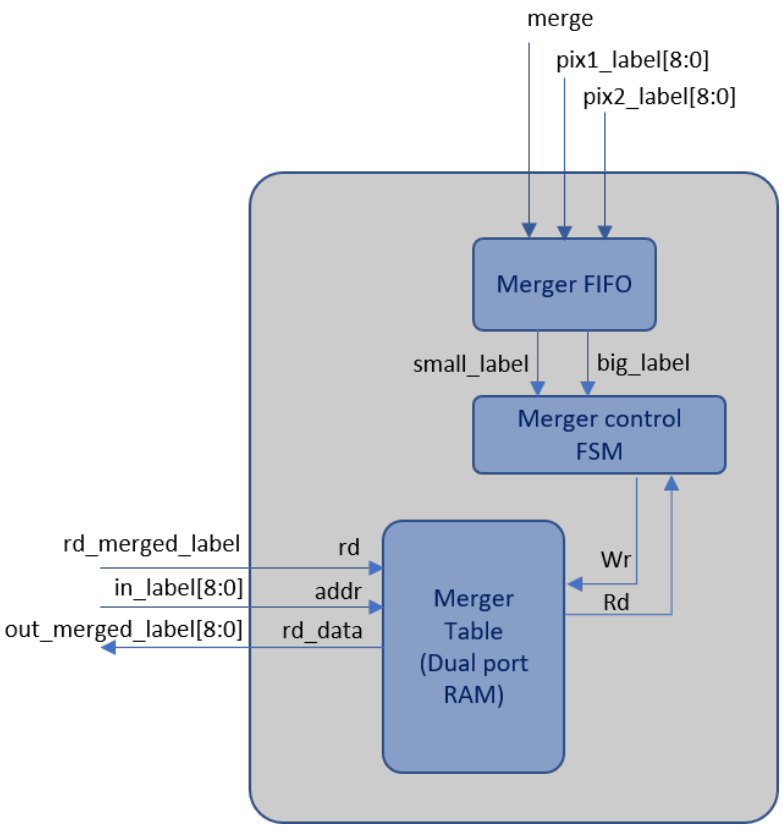

Figure 4.15: Merger Table block

Figure 4.16 shows the waveform depicting the writing of one equivalent label to the merger table.

Figure 4.16: Merger Table Waveforms

4.3.3.2 Partitioning controller

This FSM controls the communication between the Pixel FIFO, the Merger Table and the ConvertToShapeBits blocks (these blocks are shown in figure 4.12). It is responsible for segregating incoming pixels to their respective clusters based on their merged labels.

Figure 4.17: Dynamically partitioned image based on empty double columns

The sensor image is dynamically partitioned based on empty double columns as depicted in figure 4.17. A 'Double column gap detection block' detects an empty double column. When pixels for a region are input to the 'ShapeId_find' block, they are stored in the 'Pixel FIFO' until label equivalence chains are resolved for that region in the 'Merger Table'. For example, when label equivalences from Region 1 are being resolved, pixels from that region are stored in the FIFO. 'Merger Table' then proceeds to resolve equivalences of Region 2. At this time, the pixels belonging to Region1 in 'Pixel FIFO' can be read out. The partitioning controller FSM (figure 4.18) reads the pixels from the 'Pixel FIFO', reads their merged label from the 'Merger Table' and

sends the pixel to the appropriate 'Convert To ShapeBits' block based on the label using a DMUX as shown in figure 4.12.

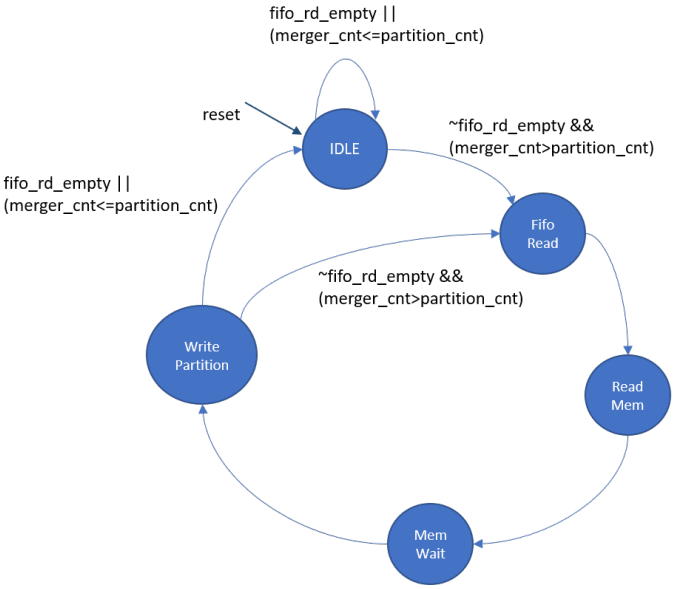

Figure 4.18: Partitioning controller FSM

4.3.3.3 ConvertToShapeBits

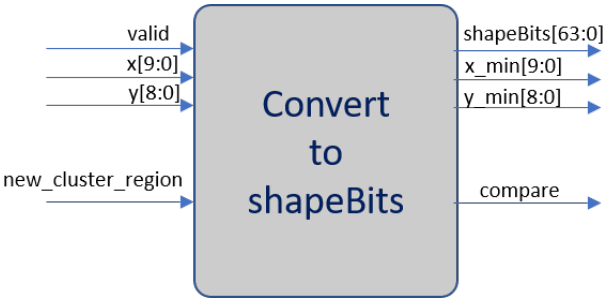

Figure 4.19: Convert To ShapeBits block

This block takes as input pixels belonging to a cluster and identifies shape bits corresponding to it for an 8x8 block. Figure 4.20 shows one such cluster and the associated shapeBit vector. Here, bit 0 corresponds to the pixel (x_min,y_min).

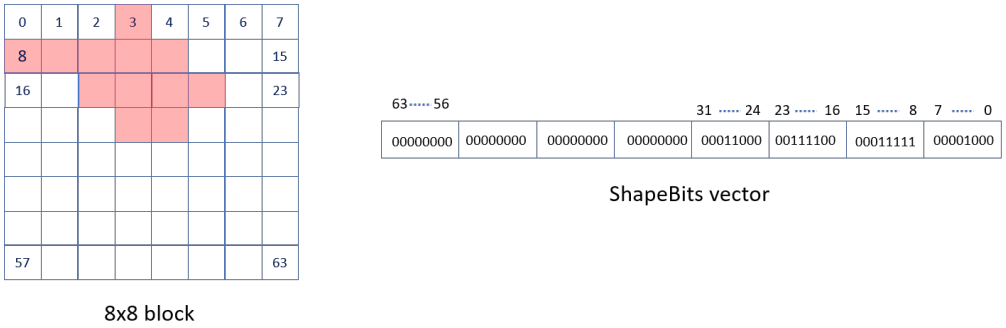

Figure 4.20: ShapeBits encoding for an 8x8 block

In the software implementation, (x_min,y_min) is found by iterating over all the pixels in the cluster first. The ShapeBits are assigned with respect to this value of (x_min,y_min) by again iterating over all the pixels. If we were to implement in the same way in hardware, we would need to store all these pixels in memory (64x19 bits). This will also add to the latency. Instead, we find the (x_min,y_min) on-the-fly and adjust the shapeBits accordingly. We use a 64-bit shift register for the shapeBits. We assume the first incoming pixel to be (x_min,y_min) and assign bit 0 of the shift register to be 1. The pixels arriving later could be greater than (x_min,y_min) or less than (x_min,y_min) in which case (x_min,y_min) needs to be updated and the shapeBits needs to be shifted appropriately. The following cases arise.

- Incoming (x,y) is greater than (x_min,y_min): (x_min,y_min) remains the same. Find relative position of (x,y) w.r.t (x_min,y_min) and set that bit to 1.

- Incoming x is smaller than x_min: x now becomes the x_min. Shift 'shapeBits' by amount (x_min-x) and set bit 0 to 1.

- Incoming y is smaller than y_min: y now becomes the y_min. Shift 'shapeBits' by amount 8*(y_min-y) and set bit 0 to 1.

The case where both x & y are smaller than x_min & y_min does not occur as the pixels are arriving in a snake pattern.

The signal 'new_cluster_region' indicates a double column gap which guarantees that the cluster region has finished. This is used to output a 'compare' signal which indicates to the next block that 'shapeBits' are now ready and can be used to lookup the dictionary.

Figure 4.21 shows the waveforms for this block while identifying the shapeBits for a sample cluster.

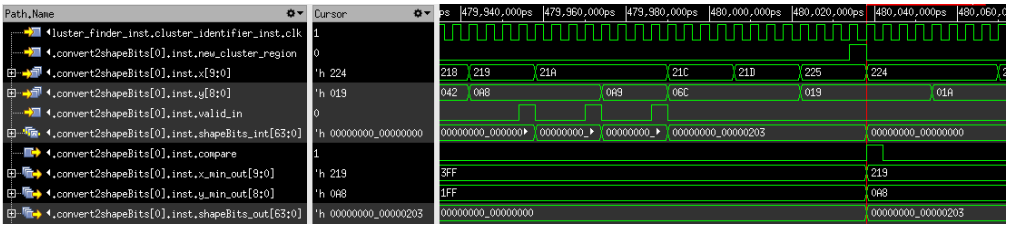

Figure 4.21: Convert to shapeBits waveforms

4.3.3.4 Dictionary

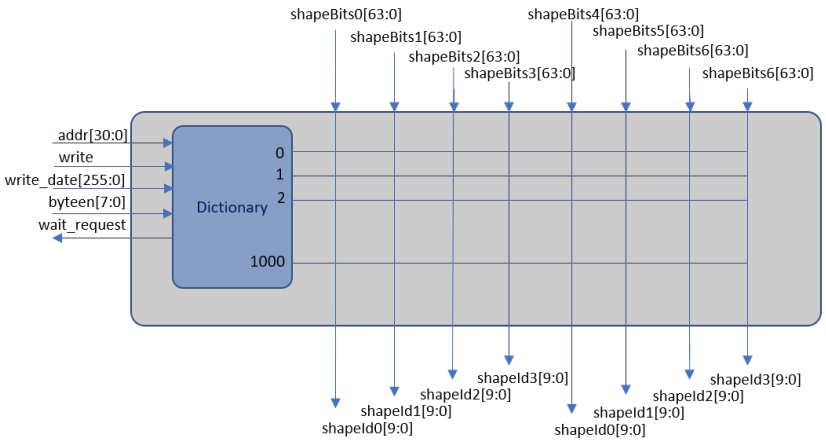

Figure 4.22: Dictionary block

The dictionary data are sent to this block by the FLP over the PCIe interface. This is stored in the dictionary memory. 'ShapeBits' along with a 'compare' signal are sent to this block from 8 'ConvertToShapeBits' blocks. These comparisons are done in parallel and the corresponding 'shapeId' which is the address of the memory corresponding to the 'ShapeBits' are sent to the output. These outputs are stored in a FIFO to be read out by the 'Output Arbiter' discussed in section 4.3.3.5

4.3.3.5 Output Arbiter

This logic arbitrates between the 32 dictionary output FIFO's in a round robin fashion and sends the corresponding cluster information (x_min, y_min, shapeId) to the output.

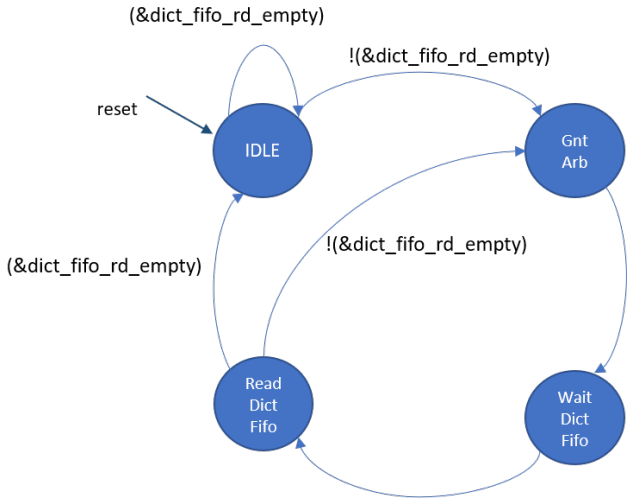

Figure 4.23: Output controller FSM

The output controller FSM shown in figure 4.23 controls the reading of dictionary output FIFOs based on arbiter grants. It transitions from IDLE to Gnt_Arb state when at least one FIFO is not empty. Based on the arbiter grant, it issues a read request to the appropriate FIFO. It waits in the Wait_dict_fifo state for the FIFO output and sends this to the output port.

4.4 Memory requirement estimate

The tables in figure 4.24 show the estimate of memory requirements for the design implementation discussed in this chapter.

Block	Feature	Memory (Bytes)	Total (Bytes)
Run-length decoder	FIFO	512	512
Cluster Identifier	Column buf0	576	1152
	Column buf1	576	
ShapeId finder	Input FIFO	464	33945
	Merger_table_fifo	9	
	Merger_table_mem	576	
	Hit to shape convertor	128	
	Dictionory	32768	
Total memory requirement/processing module			35609

Processing modules per CRU	8
Memory/processing module	35609
Memory per CRU	**284872**

Figure 4.24: Memory requirement estimate

Chapter 5

Results and Discussion

5.1 RTL level verification

The RTL design was simulated with the NCsim simulator from Cadence. The testing was done hierarchically by simulating the sub-modules of 'Cluster Finder' with a dedicated test bench.

For verifying the 'Cluster Finder' at top-level, the test bench shown in figure 5.1 was designed. For the purpose of this simulation, sensor data were generated by Monte Carlo simulations for one event. These data are read from file and fed to the DUT (Design Under Test) by the AvalonMM driver.

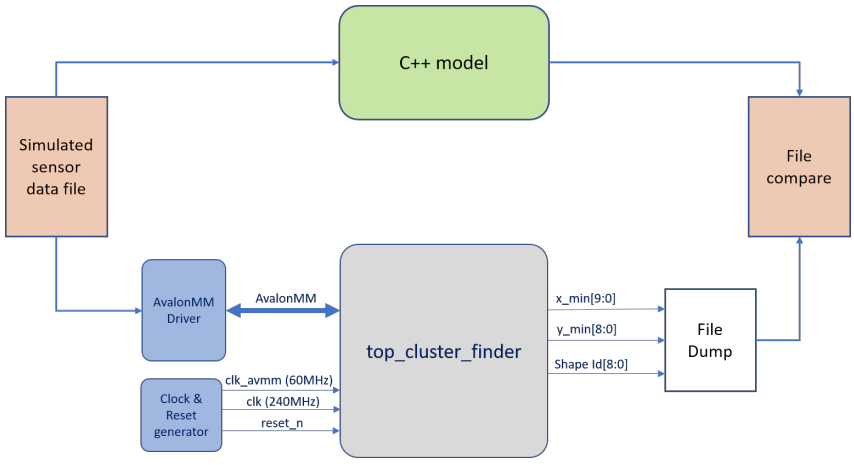

Figure 5.1: Top-level test bench

The DUT output was compared with the output generated from the C++ model and was found to match. The top-level simulation waveform in figure 5.2 shows the processing of a sample pixel, (20, 318). The latency through each sub-block is summarized in table 5.1.

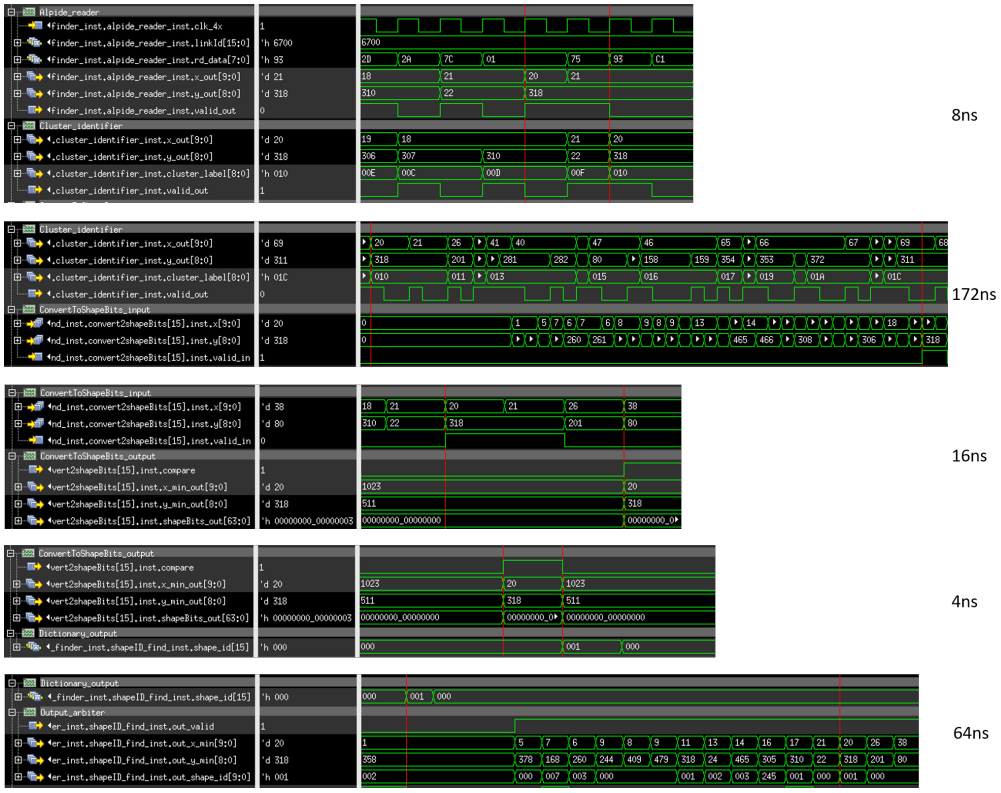

Figure 5.2: Waveforms from top-level simulations

Block	Latency (ns)
Cluster Identifier	8
Partitioning controller	172
Cluster to shapeBits	16
Dictionary	4
Output Arbiter	64
Total	**264**

Table 5.1: Latency for pixel (20, 318)

The latency of the Cluster Identifier block is constant. The latency through the partitioning block is variable and depends on the dynamic partition of the image (discussed in section 4.3.3.2). The latency through the Convert2shapeBits block is also variable and depends on the number of pixels in the dynamic partition. The dictionary has a constant latency. The time taken by each shapeID to appear at the output depends on its position in the arbitration cycle.

Software implementation	**Hardware implementation**
22.4us	3.77us

Table 5.2: Time taken to process all pixels from 1 ALPIDE sensor

Table 5.2 gives a comparison of the time taken for processing all the pixels from 1 ALPIDE sensor. We achieve 5.94X speedup compared to software by implementing the cluster finding algorithm in hardware.

5.2 FPGA resource requirements

The 'Cluster Finder' was synthesized with Intel's Quartus 17.1 tool to analyze the resources needed to implement this design on the Arria 10 FPGA.

To send the simulated sensor data to the FPGA we need a Linux host PC that can send transactions to the FPGA over the PCIe interface. Inside the FPGA, a PCIe hard IP receives these data and sends them to the Cluster Finder. To achieve this, the 'Cluster Finder' was integrated with the PCIe reference design [17] from Intel with the help of the 'Platform Designer' tool,

known as 'QSys' in the previous versions of the tool. Figure 5.3 shows the block diagram of the integrated system for FPGA synthesis and figure 5.4 shows the 'Platform Designer' window for it.

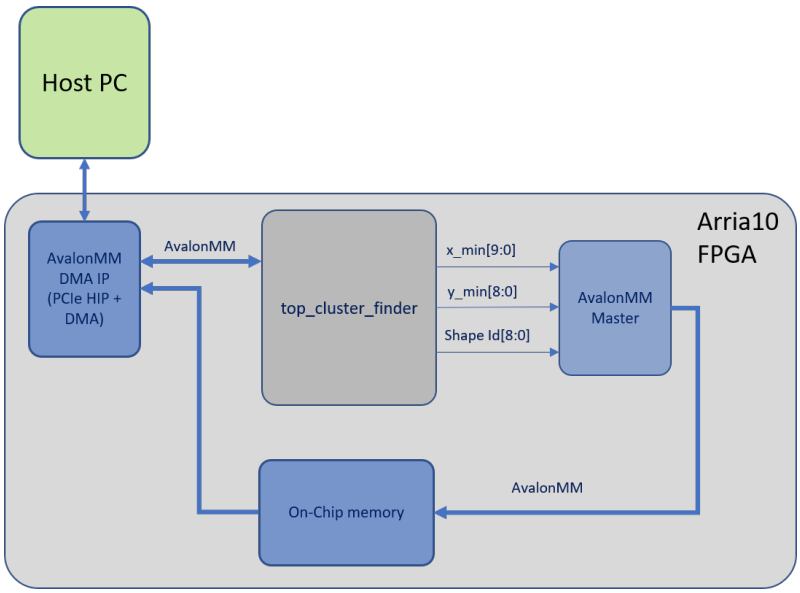

Figure 5.3: Integration of 'Cluster Finder' with PCIe reference design

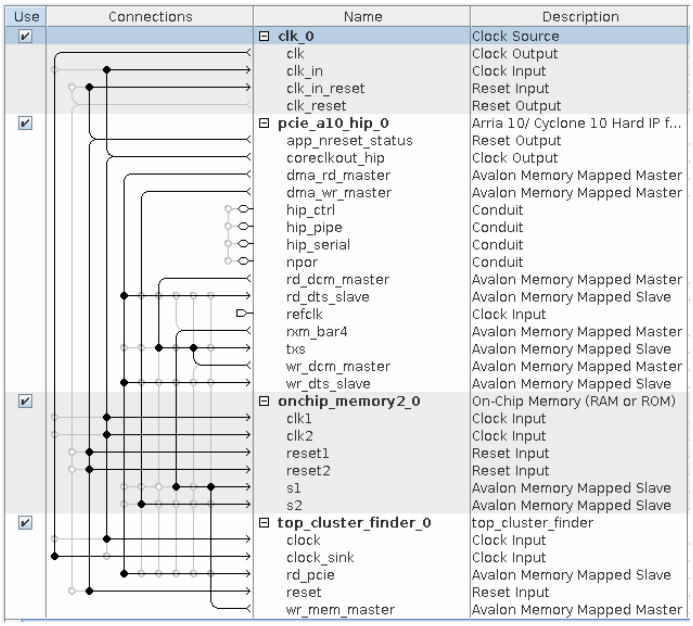

Figure 5.4: System Integration using the 'Platform Designer' tool

The synthesis results obtained for this design are given in table 5.3.

Entity	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits
alpide_reader	**496**	**96**	**4096**
cluster_identifier	**4448**	**9340**	**0**
shapeID_find	**1495248**	**268997**	**12104**
pixel_fifo	30	23	3712
merger_table	35	34	4680
partition_controller	27	58	0
convert_to_shapeBits	32x462	32x168	0
dictionary	4x369896	4x65624	0
dict_out _fifo	32x6	32x8	32x116
output_arbiter	146	66	0

Table 5.3: Resource usage from Synthesis

The resource utilization for 'alpide_reader' and 'cluster_identifier' are as per expectations. However, the logic utilization for 'shapeID_find' block is high. The major contributor to this is the 'dictionary' block. As implemented,

its memory usage is '0'. But according to our expectations, it should have used memory instead of logic. Arria 10 does not have Embedded System Blocks (ESBs) which could be used to implement the Content Addressable Memory (CAM) needed for our application. Besides ESBs, CAM can be implemented as a register array or using RAM. For implementing CAM using RAM, the role of address and data are reversed [18]. In our case, we have 64 bit dictionary entries; this will require RAM of size 2^{64}, which is not feasible. Because of this, the CAM was implemented by the synthesis tool using register array (≈65,536 registers for each dictionary) and a large amount of combinational logic. Based on current results, the design cannot be directly implemented on the Arria 10 FPGA, the bottleneck being the dictionary block. To reduce the resource utilization of the Cluster Finder, the following options could be considered:

1. Coding style: LUT usage depends on the coding style. Code optimizations [19] can result in slight reduction of resource usage.

2. Architectural changes: Architectural updates can help in reducing the resource usage significantly. The main bottleneck is the dictionary block. We use four of these blocks and in each block, we do comparison for 8 inputs in parallel. If these comparisons could be serialized, we will need only one dictionary block. Also, the combinatorial logic could be reduced.

5.3 General guidelines for hardware implementation

Based on our work, here are a set of general guidelines that can be useful while designing hardware from a software model.

- 'for' loops with 'if else' conditions within them can be designed as a state machine with the 'if else' conditions defining the state transitions.
- Bitwise operations such as the ones shown in the code snippet below can be implemented as combinational logic.

```
int y=(offsetId)/2;
int x=dColId*2 + (offsetId & 0x1) ^ ((offsetId & 0x2) >> 1);
```

- Every 'function' in C++ can be viewed as a hardware 'block' with inputs coming in and being processed every cycle.
- While implementing in hardware, combine the functionality of two loops that use the same data. With this, data need not be stored, thus, reducing memory usage as well as latency.

```
for(auto hit: cluster.hits) {
  if (hit.X<xmin) xmin=hit.X;
  if (hit.X>xmax) xmax=hit.X;
  if (hit.Y<ymin) ymin=hit.Y;
  if (hit.Y>ymax) ymax=hit.Y;
}

// bit-encode shape
shape.shapeBits=0;
for(auto hit: cluster.hits) {
  int x=hit.X-xmin;
  int y=hit.Y-ymin;
  shape.shapeBits[x+y*ClusterShape::shape1Dsize]=1;
}
```

For example, in the code snippet above, in the first iteration, minimum/maximum values of x, y are calculated. Later, the same hit data is iterated over again to calculate shapeBits. In our hardware implementation, we combined both these tasks and calculated min values of x, y on-the-fly as was discussed in section 4.3.3.3.

Chapter 6

Conclusion and Future Improvements

The software algorithm for cluster finding was successfully mapped to a Verilog implementation. The single pass implementation (not requiring a frame buffer) and efficient merger table reduced the processing latency considerably. Label merger and cluster segregation on-the-fly significantly reduced the memory requirements.

The block requiring the maximum FPGA resources is the 'dictionary'. Due to the unavailability of ESBs and the dictionary size, the synthesis tool used a large amount of logic resources to implement the CAM, overshooting the available resources of the Arria 10 FPGA. Further optimization of the Verilog code may help in reducing the logic usage, but cannot be guaranteed. Architectural changes can also be looked at.

The future steps for deploying this design on the real system would be to provide an interface that can be integrated with the 'User Logic' in the CRU. This design can be verified with the help of a CRU test bench, once it is available. In the actual system, there will be more than 24x8 instances of this Cluster Finder running in parallel, catering to the data from different sensor chips. The sensors should be evenly assigned to the different instances of the

Cluster Finder so that each has to process equivalent amounts of data and thus achieve the best overall throughput. This work was an initial development for evaluation purposes. The Cluster Finder itself can be further optimized to get the best resource usage possible.

Appendix

Appendix 1

Verilog codes for the 'Cluster Finder'

The Verilog source codes for the 'Cluster Finder' can be found at: "`https://github.austin.utexas.edu/aq3289/CRU_cluster_finder`".

Bibliography

[1] K. Aamodt, A. A. Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G. A. Rinella, Z. Ahammed *et al.*, "The ALICE experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, no. 08, p. S08002, 2008.

[2] B. Abelev, A. collaboration *et al.*, "Upgrade of the ALICE experiment: letter of intent," *Journal of Physics G: Nuclear and Particle Physics*, vol. 41, no. 8, p. 087001, 2014.

[3] S. Chapeland, B. Changaival, and T. Achalakul, "Benchmarks based on a pixel cluster finder algorithm for the future ALICE online computing upgrade," in *Real Time Conference (RT), 2014 19th IEEE-NPSS*. IEEE, 2014, pp. 1–6.

[4] ALICE Collaboration *et al.*, "Technical Design Report for the Upgrade of the Inner Tracking System," *ALICE-UGTDR1, Dec*, 2013.

[5] *ALPIDE Operations Manual*, 2016.

[6] ALICE Collaboration, "Upgrade of the ALICE Read-out & Trigger System," CERN, Tech. Rep., 2014.

[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[8] H. Flatt, S. Blume, S. Hesselbarth, T. Schunemann, and P. Pirsch, "A parallel hardware architecture for connected component labeling based on fast label merging," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on.* IEEE, 2008, pp. 144–149.

[9] Defining connectivity. [Online]. Available: http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/connect.html

[10] S.-W. Yang, M.-H. Sheu, H.-H. Wu, H.-E. Chien, P.-K. Weng, and Y.-Y. Wu, "Vlsi architecture design for a fast parallel label assignment in binary image," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on.* IEEE, 2005, pp. 2393–2396.

[11] C. T. Johnston and D. G. Bailey, "Fpga implementation of a single pass connected components algorithm," in *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on.* IEEE, 2008, pp. 228–231.

[12] M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, and S. Simon, "A memory-efficient parallel single pass architecture for connected component labeling of streamed images," in *Field-Programmable Technology (FPT), 2012 International Conference on.* IEEE, 2012, pp. 159–165.

[13] M. J. Klaiber, D. G. Bailey, Y. O. Baroud, and S. Simon, "A resource-efficient hardware architecture for connected component analysis," *IEEE*

Transactions on Circuits and Systems for Video Technology, vol. 26, no. 7, pp. 1334–1349, 2016.

[14] Aliroot tutorial. [Online]. Available: http://alice-offline.web.cern.ch/sites/alice-offline.web.cern.ch/files/uploads/OfflineTutorial/ALICE-Tutorial-Part1.pdf

[15] *Avalon Interface Specifications*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf

[16] *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_ram_rom.pdf

[17] PCI Express Avalon-MM DMA Reference Design. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/an/an690.pdf

[18] *Advanced Synthesis Cookbook*. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/stx_cookbook.pdf

[19] Minimizing FPGA Resource Utilization. [Online]. Available: http://zipcpu.com/blog/2017/06/12/minimizing-luts.html

[20] L. Musa, "Conceptual design report for the upgrade of the ALICE ITS," Tech. Rep., 2012.

[21] I. S. Association *et al.*, "IEEE Standard for Verilog Hardware Description Language. Design Automation Standards Committee, 2005," *IEEE Std 1364TM-2005*.

[22] *Intel Arria 10 Core Fabric and General Purpose I/Os Handbook*. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/a10_handbook.pdf

[23] *Arria 10 SoC Development Kit User Guide*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug-a10-soc-devkit.pdf

[24] *SCFIFO and DCFIFO IP Cores User Guide*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_fifo.pdf

[25] D. G. Bailey and C. T. Johnston, "Single pass connected components analysis," in *Proceedings of image and vision computing New Zealand*, 2007, pp. 282–287.

[26] H. M. Alnuweiri and V. K. Prasanna, "Parallel architectures and algorithms for image component labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 10, pp. 1014–1034, 1992.

[27] C. Grana, D. Borghesani, P. Santinelli, and R. Cucchiara, "High performance connected components labeling on fpga," in *Database and Expert Systems Applications (DEXA), 2010 Workshop on*. IEEE, 2010, pp. 221–225.

[28] G. Grastveit, J. Nystrand, H. Tilsner, B. Skaali, H. Helstrup, C. Loizides, T. Steinbeck, T. Alt, D. Röhrich, A. Vestbø *et al.*, "Fpga coprocessor for high-level trigger applications," 2003.